Presented to the **Software Technology Department**

De La Salle University - Manila

Term 1, A.Y. **2023-2024**

In partial fulfillment

of the course

In **CSOPESY** (**S13**)

MP2 Documentation: Process Synchronization of Looking For Group (LFG) Dungeon Queueing of an MMORPG

Submitted by:

Balcueva, Joshua Chupungco

La'O, Erin Denise Cochiengco

Sy, Matthew Jericho Go

Submitted to:

Sir Jonathan Mantua

December 4, 2023

I. Synchronization Technique

The group used a Mutex lock as the synchronization technique in the solution as shown with *acquire()* and *release()* functions being called inside a *while True* loop.

```
def instance_manager(instance_id):
      …
      while True:
            lock.acquire()
                  [Critical section]
            lock.release()
```

II. Synchronization Variables

This section lists down the variables essential for synchronizing the threads involved in the Looking For Group (LFG) process.

- **lock** - is a *threading.Lock()* object that protects the three (3) critical section variables *tank_q*, *healer_q*, and *dps_q* limiting access to these one (1) thread at a time to avoid race conditions. The *acquire()* and *release()* functions are called to set and unset this lock variable.

```
lock.acquire()
lock.release()
```

- **tank_q -** is a queue variable that pertains to the players queueing up for the Tank role. The maximum size is set to **t**, which is entered by the user.

```
tank_q = Queue(maxsize=t)
```

- **healer_q -** is a queue variable that pertains to the players queueing up for the Healer role. The maximum size is set to **h**, which is entered by the user.

```
healer_q = Queue(maxsize=h)
```

- **dps_q -** is a queue variable that pertains to the players queueing up for the DPS role. The maximum size is set to **d**, which is entered by the user.

```
dps_q = Queue(maxsize=d)
```

III. Constraints

Constraint Specifications

The group designed the code with certain constraints given to them. The following are the project constraints included in the specifications:

A) There are only n instances that can be concurrently active.

B) A standard party of 5 is 1 tank, 1 healer, and 3 DPS.

C) The solution should not result in a deadlock.

D) The solution should not result in starvation.

E) It is assumed that the input values arrived at the same time.

F) A time value (in seconds) t is randomly selected between t1 and t2. Where t1 represents the fastest clear time of a dungeon instance and t2 is the slowest clear time of a dungeon instance. For ease of testing t2 <= 15.

Output Specifications

Additionally, the outputs are also required to be in a specific format to know how well the system has performed while arranging players into parties. These outputs must be:

A) Current status of all available instances

B) If there is a party in the instance, the status should say "active"

C) If the instance is empty, the status should say "empty"

D) There should be a summary of how many parties an instance has served and total time served at the end of the execution

Code

This section describes and explains the parts of the code that were designed to satisfy these constraints.

1. To satisfy constraint A where only n instances can be active, the following list comprehension is used to instantiate n *threading.Thread()* objects and put all those instances in a list. Then using a for-loop, the list of instances (*instance_threads*) is iterated to start all the threads in the list of n length.

```
instance_threads = [threading.Thread(target=instance_manager, args=(i,))
for i in range(n)]
for thread in instance_threads:
    thread.start()
```

2. There is an *if-statement* as soon as the *while* loop starts to check if the queues for all the roles are not enough to form a standard party as defined by constraint B. If this returns *True*, then the lock is released and the *break* function will exit the loop. This ensures that constraint B is satisfied where parties can only be formed as long as there is enough remaining roles defined by having at least 1 tank, 1 healer, and 3 DPS.

```
if not all([tank_q.qsize() >= 1, healer_q.qsize() >= 1, dps_q.qsize() >=
3]):
      lock.release()
      break
```

3. If there are enough players in each queue to form a standard party, then the rest of the code within the loop will execute. The required number of members for all the roles will be obtained through the *get()* function (which also removes those items from the queue) and lock will be released.

```
tank = tank_q.get()
healer = healer_q.get()
dps_list = [dps_q.get() for _ in range(3)]
lock.release()
```

4. Deadlock is only possible if a circular wait is present. In our implementation, since there is no potential for circular waits to happen, deadlock is avoided satisfying constraint C.

5. In satisfying constraint D, it is presumed that starvation can be more likely at low values of t1 and t2, especially when it is equal to zero (0) as threads may sleep too short and reenter their own critical section again before other threads can do the same. In practice, this was not observable in our Python implementation.

6. To fulfill constraint F, this section of the code simulates the time that the players are clearing the dungeon, with the variable **dungeon_time** being a random integer between t1 being the minimum time and t2 being the maximum time.

```
dungeon_time = random.randint(t1, t2)
time.sleep(dungeon_time)
```

7. Lastly, to fulfill the output requirements, each time the program loops, the **instances_summary** variable is updated to tally the total count of parties served and time spent in dungeon for each thread which will be printed at the end of program execution. The **logging** module was also initialized at the start of the program with a format specified, being used to print if an instance is active or empty to show the current summary of the party finding progress.

```
LOG_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %(message)s'
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
```

```
instances_summary = [{'parties_served': 0, 'time_served': 0} for _ in range(n)]
```

```
logging.info(f"Instance {instance_id+1}: active")
```

```
instances_summary[instance_id]['parties_served'] += 1
instances_summary[instance_id]['time_served']+= dungeon_time
```

```
logging.info(f"Instance {instance_id+1}: empty")
```

```
for i, summary in enumerate(instances_summary):
      print(f"Instance {i+1} served {summary['parties_served']}
      parties and was active for {summary['time_served']}
      seconds.")
```